A Syntax Reference Manual for the Euclideus Language

Justin Gagnon

April 13, 2018

This document is meant as a reference manual to assist in writing scripts and programs in the Euclideus computer language, which is based on a computer algebra system involving recursive binary search trees coupled with complex arithmetic. An initial design goal of this language is to augment the capabilities of the Euclideus system by providing to the user basic procedural programming functionality, thus enriching the types of mathematical evaluations that may be performed, while preserving the overall efficiency of the system as well as its algebraic nature. This programming language should be considered as a superset of the current features of the system, such that none of the previous functionality is rendered obsolete; rather, the programming language simply represents an augmentation of the Euclideus toolbox with respect to previous functionality. The following document serves to expose the main features and usage patterns of this language.

CONTENTS

1 OVERVIEW

The Euclideus syntax is, for the time being[1], based on a procedural paradigm. A script is viewed as a sequence of commands, each of which may be one of five basic types:

1. *Arithmetic*: An arithmetic command instructs the interpreter to evaluate a standard arithmetic expression, given with respect to the {+ - * / ^ ' !} operators – representing mathematical addition, subtraction, multiplication, division, exponentiation, complex conjugation, and the factorial[2], respectively. These commands are all assumed to produce a quantity of complex nature, i.e. with a real part and an imaginary part each in double-precision floating point form. An arithmetic expression that does not evaluate to a complex numerical result is processed using a factorization algorithm based on recursive binary search trees (rBST), and output, by default, in its factorized form. Please refer to the documentation manual in the Euclideus system for further information about arithmetic operations and the basic predefined functions that are recognized by the syntax.

2. *Boolean*: Like arithmetic commands, a Boolean command may be given as an expression in terms of Boolean arithmetic. Such expressions are assumed to be of value 0 or 1. In the present syntax, Boolean expressions are understood algebraically with respect to two classes of operators. The primary Boolean operators { | & ~ } are the inclusive OR, the AND, and the negation (NOT) operators, respectively. These three primary Boolean operators, of which | and & are infix and ~ (the tilde, representing NOT) is prefix, act on other Boolean expressions, i.e. values or expressions assumed to evaluate to 0 or 1. In addition to these primary Boolean operators, the Euclideus syntax recognizes the Boolean comparator operators { == != <= >= < > }, all of which are of infix-type. These operators act between two operands, operating only on their real parts (imaginary parts of operands are simply ignored by all Boolean comparators).

3. *Control blocks*: A control block is essentially a sequence of one or more comma-delimited instructions bracketed by the : and ; symbols. Euclideus recognizes three different types of control blocks: (i) the `if:;` block for branching, (ii) the `for:;` and `while:;` blocks for looping and (iii) the ordinary linear instruction sequence, which encloses a scope if bracketed by : and ;.

4. *Assignment*: Assignment statements are those of the type <quantity>=<value>. In such types of statements, the left-hand-side ("quantity") is a placeholder-name of a variable or function into which the "value" is to be stored/inserted. Currently, Euclideus supports assignments to variables and to functions. In a valid assignment statement, the left-hand-side contains none of the aforementioned operators, arithmetic nor Boolean, nor should it contain bracketing symbols (unless it is a function signature e.g. `f(x,y,z)`);

[1] The inclusion of classes will later be implemented, as well as more complex data types such as arrays, lists, trees, etc.

[2] The factorial operator actually operates on a complex-valued operand because it is implemented using the Gamma function.

the right-hand-side is a properly formed Boolean or arithmetic expression, or a control block if the left-hand-side of the assignment is a function signature.

5. *Reserved keywords*: Certain instruction keywords are reserved for use within specific contexts and control blocks. These include `break`, as well as `elseif`, and `else` specifically for use within branching statements.

All scripts in Euclideus are composed of sequences of commands that are one of the above five types. For further documentation on arithmetic operations and assignment operations involving variables, functions and simple arithmetic expressions, please refer to the Euclideus system's documentation manual.

2 Boolean Operations

2.1 The Primary Operators: Negation "~", Conjunction "&", and Disjunction "|"

The implementation of Boolean math is based on the standard Boolean arithmetic using the three primary Boolean operators: OR (`|`), AND (`&`) and NOT (`~`) bracketed by the ordinary parentheses "`(`" and "`)`", resulting in Boolean expressions assumed to take on the values `0` or `1`[3]. For example, the expression `a&~a` evaluates to `0`, and `a|0` evaluates to `a`.

The negation operator `~` takes precedence over the other two { `&` `|` }, such that for e.g. `~a&~b`, `a` and `b` are first individually negated followed by the application of the AND operation. A sequence of negations produces the expected result: `~~~a`⟶`~a`.

The AND operator takes precedence over the OR operator. Therefore, brackets must be used in order to ensure that a disjunction is evaluated before a conjunction. For example, the expression `a|b&c` is interpreted internally as `a|(b&c)`. If the OR is intended to take precedence over the AND, this expression must be input as `(a|b)&c`.

2.2 The Numerical Comparators: ==, ~=, <, <=, > and >=

The Euclideus syntax supports the standard comparison operators found in most other languages: "is equal to" (`==`), "is not equal to" (`~=`), "is less than" (`<`), "is less than or equals to" (`<=`), "is greater than" (`>`) and "is greater than or equals to" (`>=`). These comparators act between two numerical/arithmetic operands and ignore their imaginary parts while operating only on their real parts. Thus, `3>2+100*i` evaluates to `1` despite the RHS of the comparison being much larger in magnitude. To compare imaginary parts instead of real parts, one may use the `im()` function, i.e. `im(3)>im(2+100*i)` then evaluates to `0`.

Note that all these comparison operators take precedence over the primary Boolean operators { `~` `|` `&`}. Thus for example, the expression `~3<2|a&~a` is the same as `~(3<2)|(a&~a)` ⟶

[3] Since Boolean results are numerical, they can actually be used as part of the arithmetic syntax, e.g. `sin(a|~a)` is understood as `sin(1)`.

`~0|0` ⟶ `1|0` ⟶ `1`.

As a final remark on Boolean expressions, Boolean expressions that do not evaluate to `0` or `1` – which also includes any number other than `0` or `1` – are simply taken as indeterminate. For example, an expression like `a|~3.3` would not be further processed by the Boolean evaluator.

3 Control Blocks

For control blocks, the Euclideus syntax assumes bracketing using the colon ":" as the opening bracket and the semicolon ";" as the closing bracket, and the comma "," for delimitation between commands. These opening and closing brackets define scopes for the set of assigned variables and functions that may be accessed and modified inside the control block.

After the interpreter has completed processing all the instructions by reaching the end of a control block, the scope defined by the ":" and ";" enclosing brackets ends, which means that any variable or function assignments *created inside the scope* of the control block are local, i.e. they are erased from memory as soon as the control block ends. On the other hand, changes made to any pre-existing variable or function assignments are preserved in the outer scope upon exiting the block.

These scoping characteristics hold true for all three types of control structures discussed below.

3.1 The Linear Command Sequence

As its name implies, the Linear Command Sequence is the simplest of the control blocks as it merely consists of a sequence of one or more comma-separated commands, to be evaluated linearly from the first command to the last one in the sequence. As an example, consider the following:

```
:a=3, b=sqrt(-a), a^b/c;
```
(3.1)

This statement can be understood as follows: (I) a variable `a` is first assigned the value of 3, (II) followed by a variable `b` assigned the value of `sqrt(-a)`, (III) followed by the evaluation of the expression `a^b/c`. Assuming `c` was never defined, it remains undetermined. The first two instructions are assignments and therefore do not produce any output, while the third instruction is of arithmetic type and therefore tells the interpreter to evaluate the expression, resulting in an output of `(-0.326+0.9454*i)/c`. If more than one such output expression is requested in a given command sequence, the interpreter simply concatenates all the outputs it encounters into a comma-separated list.

Within the local scope of the command sequence, all variables and functions previously assigned in the outer scope are accessible. Thus, if `c` had already been defined outside of the

scope of the command sequence, say as `3.1416`, it will be understood as `3.1416` within the scope of the command sequence which means that the third instruction will produce the result `-0.1038+0.3009*i` (that is, with `c=3.1416`). The variable `c` may also be redefined inside the command sequence.

Unlike the other types of control blocks, the linear command sequence also allows to process a set of instructions without defining a new scope, i.e. by omitting the ":" and ";" brackets. In this case, no local scope is defined and all variables and functions created in the command sequence remain stored in memory.

3.2 The Branching Statement

The branching block is defined by the "if" statement, it is a control block bracketed by "`if:`" and "`;`", which also encloses a sequence of commands. The branching statement may be viewed as a generalization of the simple linear command sequence as it is essentially a *decision tree* of linear command sequences, using the special keyword commands "`elseif`" and "`else`", which are contained *inside* the branching block.

The following example is representative of the syntax of a general branching block:

```
if:x<0,                                    (3.2)
   y=1/x,                                  (3.3)
   elseif, x>=100 && x<=1000,              (3.4)
     z=ln(x),                              (3.5)
     y=3^z,                                (3.6)
   elseif, x<100,                          (3.7)
     y=3-x,                                (3.8)
   else,                                   (3.9)
     z=exp(-x),                            (3.10)
     y=sin(z);                             (3.11)
```

First, notice that the entire block is bracketed by the colon immediately after the "`if`" word, and it is closed after the command sequence following the "`else`", with a semi-colon at the end of line (3.11). The interpreter ignores any white space or new line characters, therefore the closing semi-colon bracket may be placed on its own line, and individual commands of a sequence may be juxtaposed on the same line, and indentation is arbitrary. Note also that a valid Boolean expression must necessarily be the first command immediately following the "`if:`" opening bracket and valid Boolean expressions must always follow any "`elseif`" keywords. The "`elseif`" together with the "`else`" keywords are optional.

The above branching block encloses a total of four command sequences, however, referring to Fig-3.1, *at most* one of these will be executed depending on the truth-value of the open-

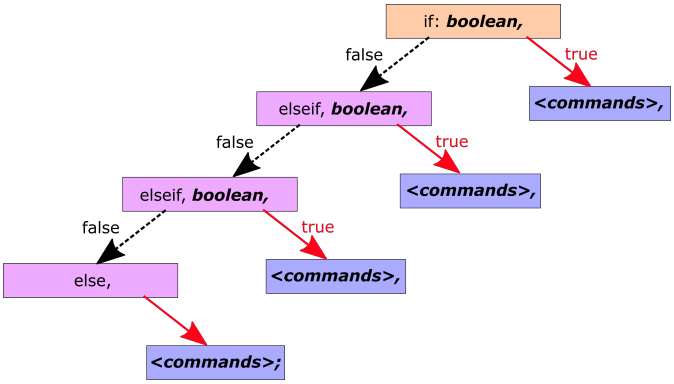

Figure 3.1: A schematic of the general branching control block. The "`if`" is the root of the decision tree and the "`else`" and "`elseif`" statements are reserved keywords inside the control block. The dotted lines leading to the "`else`" and "`elseif`" statements are to indicate that they are optional.

ing Boolean expressions, i.e. the Boolean expression immediately following the "`if:`" or an "`elseif`". Once an opening Boolean expression evaluates to `1` ("true"), the remaining command sequence is executed until an "`elseif`" or an "`else`" keyword is reached[4], which signals an exit of the branching block. The command sequence following "`else`" (if it exists) is only executed provided that all previous opening booleans expressions evaluate to `0` (false).

Taking the example above, only lines (3.5) and (3.6) are executed if `x` is between 100 and 1000, while only lines (3.10) and (3.11) are executed if `x` is greater than 1000.

Just as for linear command sequences, all variables and functions created within a branching block are local to the scope of that block and are erased after exiting the block, while if they were previously defined outside the scope of the block, they remained modified after the block. Thus, if `y` was never created outside of the `if:;` block, it would remain unassigned after processing that block. However, if it was previously created outside the `if:;` block, and if `x` was approximately equal to `403.428793`, it may be verified that `y` would carry a value of about `3^6` following this branching statement.

Finally, an opening Boolean expression that does not evaluate to `0` or `1` is assumed to be neither true nor false, but rather unknown. If the interpreter encounters such an undefined Boolean expression in the branching blocks, it exits the block and reports an error message associated to the undefined expression.

[4]The "`break`" keyword may also be used to suddenly exit the branching control block, however, as it is more relevant to looping structures it will be discussed in the next section.

3.3 THE LOOP STATEMENTS

The Euclideus syntax supplies two different types of loop constructs: the "while" loop and the "for" loop. Just like the previously discussed control blocks, the loop blocks enclose their own scopes in the same manner as the "`if:;`" structure and the linear command sequence. Loop blocks recognize only a single special keyword command: the "`break`" command which is used to immediately exit a loop. Its usage will be detailed at the end of this section.

3.4 THE WHILE LOOP

The simpler of the loop statements is the "`while:;`" loop block, shown in Fig-3.2a. The while-loop has a structure similar to that of the "`if:;`" block in that it opens up with a Boolean expression. This opening Boolean expression determines whether or not the interpreter enters the loop. In contrast to the branching structure, a looping structure returns to the beginning of the command sequence when it reaches the closing bracket "`;`". The following code is a typical and simple sequence of commands involving the "`while`" loop:

```
n=0,                    (3.12)
y=0,                    (3.13)
while:n<=32,            (3.14)
  y=y+x^n,              (3.15)
  n=n+1;                (3.16)
```

The first two lines are used to initialize a couple of quantities `x` and `n`, the latter of which appears in the loop's opening Boolean expression (if it were not initialized, that expression would not evaluate to `1` or `0` and hence the interpreter would throw an error message complaining of an undefined Boolean expression). In order for this loop to end, the Boolean expression on line (3.14) must become false at some point during the execution, which is one of the elements of the opening Boolean statement must be altered at some point during the execution of the command sequence inside the loop. In this example, an *incrementing assignment* is included on line (3.16) so that `n` may eventually become larger than `32`. Once this happens, the opening Boolean expression evaluates to `0` (false), and the loop exits its scope.

Now since both `n` and `y` were created outside of the scope of the loop, they remain modified once the loop has exited from its scope. It may be seen that `n` will carry the value of `33` after the loop, and `y` will be an algebraic polynomial of order `32` ($1+x+x^2+\ldots+x^{32}$) if no value was previously assigned to `x` outside the scope of this discussion.

3.5 THE FOR LOOP

The `for:;` block is *almost* equivalent to the `while:;` block, but ensures that the initialization and incrementing commands – referring to (3.12) and (3.16), respectively – are syntactically required, particularly at the outset of the block. The `for:;` block is a device that forces the

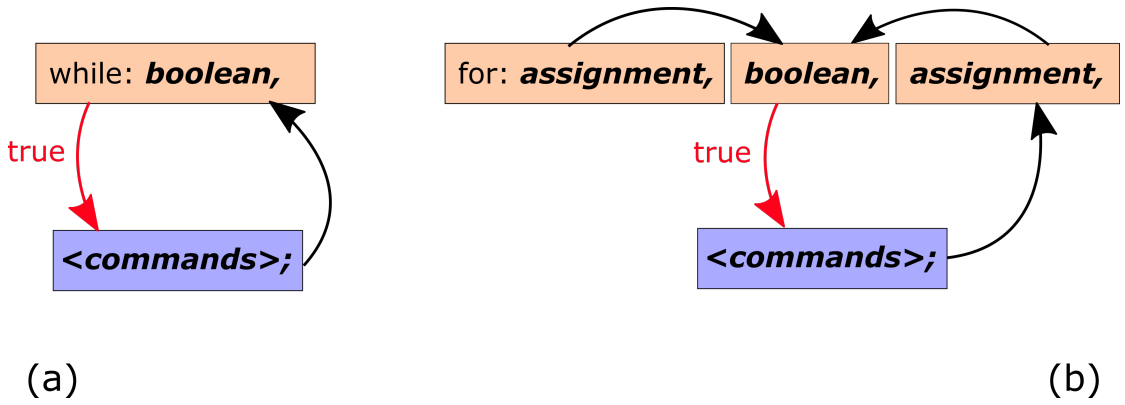

Figure 3.2: Panel-a shows the most basic of loops, the `while:;` loop. In order to ensure that the loop is not infinite, one of the instructions in the command sequence (the blue box in panel-a) must ensure that the loop's opening Boolean expression becomes false at some point during execution. Panel-b, exposes a schematic of the `for:;` loop, which allows for more explicit initialization and incrementing of a "dummy" variable, thus ensuring a safer looping block (making it more troublesome to code infinite loops).

programmer to make sure that infinite loops don't happen. Placing (3.12) as the first argument right after the opening bracket (`while:`) and placing the incrementing statement (3.16) immediately after the opening Boolean expression (should) ensure that all looping conditions are met, so as to avoid infinite loops, overflow, and so on. With the
with the help of the "`for:;`", as defined, the previous lines from (3.12) to (3.16) may instead be written as follows:

```
y=0,                                  (3.17)
for:n=0,n<=32,n=n+1,                  (3.18)
  y=y+x^n;                            (3.19)
```

As shown on line (3.17), the syntax of the "`for:;`" loop mandates the definition of the loop conditions from the outset. These conditions are: (i) an initialization statement for the loop's iterator (`n=0` in the above example), (ii) a condition for ending the loop, expressed as a Boolean expression (`n<=32`) and finally (iii) an assignment statement which serves to update the iterator (`n=n+1`); the third statement (iii) being executed *a the end* of the loop, i.e. after the last command in the sequence enclosed by the `for:;` block. A flowchart of the `for:;` block is displayed in Fig-3.2b.

An important difference between the `for:;` and `while:;` blocks is that the iterator of the `for:;` block is by default local to its scope[5]. Thus, in the above example, notice that `n` is

[5]In the case of the `while:;` block, any iterator used in a condition for stopping the loop must necessarily be

assigned after the "`for:`" opening bracket, which is inside the scope of the `for:;` block. If `n` had not been previously assigned outside of the scope of the `for:;`, it will remain unassigned once the interpreter has completed execution of the "for" loop.

3.6 The Break Command

In addition to the Boolean condition, loop structures may be halted abruptly by making use of the `break` keyword. Once the interpreter encounters the `break` command at any point within the execution of the loop, it immediately exits the loop and continues execution of commands beginning immediately following the loop block. If the `break` command is placed inside a series of nested branching blocks, it also immediately halts execution of all `if:;` blocks that enclose the `break` keyword until it finds an enclosing loop block, at which point the interpreter exits the loop successfully (i.e. without throwing an error message). The `break` key word essentially "bubbles up" through all enclosing branching blocks until an enclosing loop is found, at which point the bubble pops and the remaining commands are executed as if the loop exited normally. Please consider the following as a general usage example of the `break` command:

```
sum=0,                                  (3.20)
N=0,                                    (3.21)
y=0,                                    (3.22)
for:n=1,0==0,n=n+1,                     (3.23)
  sum=sum+1/n,                          (3.24)
  if:sum>10,                            (3.25)
    N=n,                                (3.26)
    y=sum^2,                            (3.27)
    break;                              (3.28)
;                                       (3.29)
```

The above example computes $\sum_{n=1} n^{-1}$ until the summation reaches a value greater than `10`, at which point it stores the number of terms evaluated in the variable `N`, the result $\left(\sum_{n=1}^{N} n^{-1}\right)^2$ into the variable `y`, and exits the loop. Notice that the second argument of the loop, `0==0`, is a tautology (it always evaluates to "true") and therefore the only command that is able to stop the execution of the `for:;` loop in this case is the `break` command. As the `break` is inside an `if:;` block, it gets relayed outside of the `if:;` and is detected in the scope of the `for:;` as a halting signal, which stops the loop execution immediately after the `if:;` block. The loop reports a successful exit upon detection of a `error`[6]. Since the variables `sum`, `y` and `N` are all

initialized before entering the "while" loop, i.e. outside of the loop's scope.

[6] A successful completion of the loop means that no further `break` statements are signalled. So, for instance if the `for:;` block in (3.20)-(3.29) was nested inside the scope of another loop, the outer loop would keep on running seeing as it would not detect any `break` command; the `break` signal only bubbles up through nested scopes until it reaches the scope of a loop block.

Figure 4.1: A function may be viewed as a black box with only two unidirectional channels for interaction with an outer scope. The input channel takes in a series of input arguments which are copied to the function's local scope, and an output is provided at the end of the function's execution.

initialized outside of the scope of the `for:;` block, they remain assigned in the outer scope.

4 Function Definitions and Execution

In ordinary algebra, a function may be viewed as a device that maps a set of input arguments, say x, y and z to an output result $f(x,y,z)$. To define such a mapping, it is sufficient to assign an algebraic expression to the function signature, by saying for example $f(x,y,z) = x^2 - y/z^3$. Such a purely algebraic definition is limited by the standard mathematical operations which do not straightforwardly include operations involving the logical control structures described in the previous section. In order to alleviate this deficiency, the Euclideus syntax provides a mechanism to define functions as more elaborate command sequences with a return value. The function's return value is expressed as an assignment to the `return` variable, a specially-recognized variable within the context of the function definition, interpreted as the result to output upon completion of the function's execution.

As an example, the factorial function (although already provided by the "`!`" operator) may be defined as a function programmatically in the following manner:

```
factorial(N)=:                    (4.1)
  return=1,                       (4.2)
  if:N<=1, break;                 (4.3)
  for:n=2,n<=N,n=n+1,             (4.4)
    return=n*return;              (4.5)
;                                 (4.6)
```

This example shows the effect of two specially recognized keywords in the context of a function definition: the `return` variable and the `break` command. Lines (4.2) and (4.5) contain assignments to the `return` variable, which is a variable local to the scope of the function, i.e. everything between the "`=:`" opening bracket all the way to the last "`;`" – the closing bracket of the function definition. Whatever value the return variable contains upon completion of

the function execution is the function's output result as returned through the output channel.

In order to have this function execute its code, the syntax is simply to spell out the function with a given input argument, e.g. `f(12)` would compute "twelve factorial". Note that if the input argument `N` is not greater than 1, the "if" statement on line (4.3) evaluates to "true" and the interpreter encounters the `break` command. Just as for loop structures, the break command is understood within the context of a function definition as a signal to leave the function. This may be warranted in cases when a value is assigned to the return variable *and* no further execution of the function is warranted. Thus, in the example above, when `N` is no larger than one, the "for" loop on (4.5)-(4.6) is never executed and the function exits, returning a value of one. If no return value is ever set during the function call, the function simply returns its own signature with input arguments, e.g. in this example it would return `f(N)` if for some reason no return were specified/assigned.

4.1 The Scope of a Function

Execution of function code occurs within a more restricted environment than for control blocks. As schematized in Fig-4.1, a function is like a black box to which is offered only two means of interaction with the outside scope. From within the function scope, all assignments made in the outer scope are visible but can only be modified inside the function's scope, which is to say that any reassignments of outside variables or functions persist only within the function's scope. In other words, the function code sees everything outside of it, but cannot make any changes to its environment.

For instance, the number `3.1416`, which is defined as the variable `pi` in the default configuration of the Euclideus environment, is accessible from within a function call, but if the variable `pi` is reassigned to a new value at some point during the function's execution, this change is local to the function's scope and `pi` remains assigned to `3.1416` in the outer scope, regardless of how the function uses the variable `pi` within its own scope. Of course, just as with control structures, any variables created within the function call (including the `return` variable) are all erased when the function exits its scope.

4.2 Recursion

Recursion is supported in the Euclideus syntax. In order to illustrate how a recursive function may be defined, the following is a recursively-defined version of the previously discussed factorial function.

```
factorialRecursive(N)=:                                  (4.7)
  if:N<=1, return=1,                                     (4.8)
  else, return=N*factorialRecursive(N-1);                (4.9)
;                                                        (4.10)
```

As is typical for recursive definitions, a base case, here (4.8), is necessary in order to avoid infinite recursion. When the interpreter launches the function's code, it first checks to see if `N` is smaller or equal to one, if so it returns one. Otherwise, it assigns a value to the return variable (4.9) which is obtained via an additional call to the `factorialRecursive` function itself. As you can see, if no base case (4.8) were ever defined, the `factorialRecursive` function would endlessly be calling itself until a stack or floating-point overflow occurred.

4.3 An Example Function that Evaluates a Composite Fourier Series

Using the present syntax, it is possible to write functions that are able to do more than just evaluate single mathematical expressions. As a case in point, let us define a function that is a Fourier series representing alternating square-wave/sawtooth-wave from cycle to cycle. Since Fourier series must be truncated at some value N in order to be computable, let us also include that cutoff integer N as a parameter in our function. Such a function may therefore be written as

```
fourSawSquare(x,N)=:                                    (4.11)
  if:mod(ceil(x/(2*pi)),2)==0,                          (4.12)
    return=0,                                           (4.13)
    for:n=1,n<=N,n=n+1,                                 (4.14)
      return=return+(-1)^(n+1)/n*sin(n*x);              (4.15)
  elseif,mod(ceil(x/(2*pi)),2)==1,                      (4.16)
    return=0,                                           (4.17)
    for:n=1,n<=N,n=n+2,                                 (4.18)
      return=return+2/n*sin(n*x);                       (4.19)
  ;                                                     (4.20)
;                                                       (4.21)
```

As is clear from this function, two different Fourier series are evaluated depending on the value of `x`. Moreover, note that it's not necessary to define `pi=3.1416` because it is already a defined quantity in the outer scope. In addition, since the modulo, the ceiling and the sine functions are already defined in the outer scope, they can also be used from within the `fourSawSquare` function. This code is called in the same manner as any other mathematical function, e.g. `fourSawSquare(2,10)` would evaluate this composite Fourier series at position `x=2` using `N=10` harmonics. Thus, a 1D plot of the function can be made for a given number N of harmonic functions. Fig-4.2 shows two plots of this function for $N = 5$ and for $N = 20$.

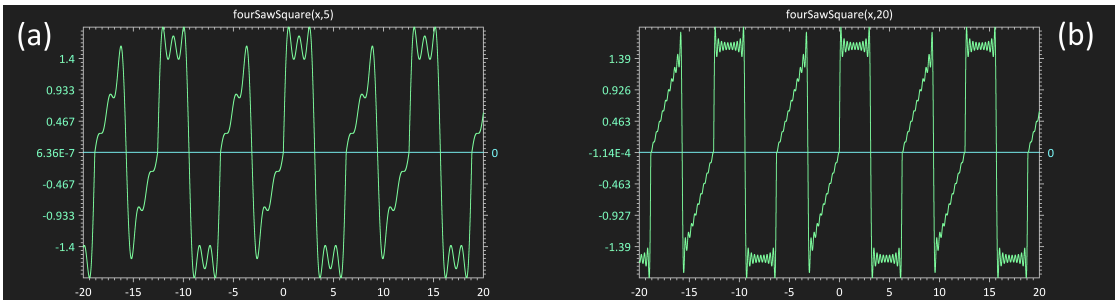

Figure 4.2: The composite Fourier series function defined in (4.11)-(4.21) is plotted between $x = -20$ to $x = 20$ using (a) $N = 5$ harmonics and (b) $N = 20$ harmonics, showing the alternating sawtooth/square waves from cycle to cycle.